Introduction to Python & Data

Research Data Management Support

2/3/23

Table of contents

Welcome!

Python is a powerful programming language that is popular can be used to write scripts for to work effectively and reproducibly with data. In this workshop, we aim to give you the tools to start exploring Python and all it has to offer by yourself.

1 Acknowledgements

Our course materials are an adaptation of the …. Carpentries course, reused under a CC-something license. The Intro to R & Data course at UU also served as a reference.

1.1 Contributors

- Jelle
- Christine
- Roel
- Neha
- Stefano

Part I

Preparation

2 Schedule

we can add a detailed schedule here later

3 Installation & Setup

3.1 Overview

The course materials are designed to be run on a personal computer. All of the software and data used are freely available online, and instructions on how to obtain them are provided below.

If you have any questions, or are not comfortable with doing the installation yourself join the RDM walk-in hours to ask for help.

3.2 Install Python

In this course, we will be using Python 3 with some of its most popular scientific libraries. Python is a popular language for research computing, and great for general-purpose programming as well. Although one can install a plain-vanilla Python and all required libraries by hand, we recommend installing Anaconda, a Python distribution that comes with everything we need for the lesson. Detailed installation instructions can be found in the Anaconda documentation, or by following the instructions below.

Regardless of how you choose to install it, **please make sure you install Python version 3.x** (e.g., 3.9 is fine).

We will teach Python using the Jupyter Notebook, a programming environment that runs in a web browser (Jupyter Notebook will be installed by Anaconda). For this to work you will need a reasonably up-to-date browser. The current versions of the Chrome, Safari and Firefox browsers are all supported (some older browsers, including Internet Explorer version 9 and below, are not).

3.2.0.1 Windows

1. Open https://www.anaconda.com/products/individual#download-section with your web browser.
2. Download the Anaconda for Windows installer with Python 3. (If you are not sure which version to choose, you probably want the 64-bit Graphical Installer `Anaconda3-...-Windows-x86_64.exe`)

3. Install Python 3 by running the Anaconda Installer, using all of the defaults for installation *except* make sure to check **Add Anaconda to my PATH environment variable**.

3.2.0.2 MacOS

1. Open https://www.anaconda.com/products/individual#download-section with your web browser.
2. Download the Anaconda Installer with Python 3 for macOS (you can either use the Graphical or the Command Line Installer).
3. Install Python 3 by running the Anaconda Installer using all of the defaults for installation.

3.2.0.3 Linux

1. Open https://www.anaconda.com/products/individual#download-section with your web browser.
2. Download the Anaconda Installer with Python 3 for Linux. (The installation requires using the shell. If you aren't comfortable doing the installation yourself stop here and come to the walk-in hours to ask for help.)
3. Open a terminal window and navigate to the directory where the executable is downloaded (e.g., `cd ~/Downloads`).
4. Type

   ```
   bash Anaconda3-
   ```

   and then press Tab to autocomplete the full file name. The name of file you just downloaded should appear.
5. Press Enter (or Return depending on your keyboard). You will follow the text-only prompts. To move through the text, press Spacebar. Type `yes` and press enter to approve the license. Press Enter (or Return) to approve the default location for the files. Type `yes` and press Enter (or Return) to prepend Anaconda to your `PATH` (this makes the Anaconda distribution the default Python).
6. Close the terminal window.

3.3 Obtain lesson materials

1. Download ??-data.zip and ??-code.zip (see Course Materials).
2. Create a folder called `intro-python` on your Desktop.
3. Move downloaded files to `intro-python`.
4. Unzip the files.

You should see two folders called `data` and `code` in the `intro-python` directory on your Desktop.

3.4 Launch Python interface

To start working with Python, we need to launch a program that will interpret and execute our Python commands. Below we list several options. If you don't have a preference, proceed with the top option in the list that is available on your machine. Otherwise, you may use any interface you like.

3.5 Option A: Jupyter Notebook

A Jupyter Notebook provides a browser-based interface for working with Python. If you installed Anaconda, you can launch a notebook in two ways:

3.5.0.1 Command line (Terminal)

1. Navigate to the `data` directory: **Unix shell** If you're using a Unix shell application, such as Terminal app in macOS, Console or Terminal in Linux, or Git Bash on Windows, execute the following command:

   ```
   cd ~/Desktop/intro-python/data
   ```

   Command Prompt (Windows) On Windows, you can use its native Command Prompt program. The easiest way to start it up is pressing Windows Logo Key+R, entering `cmd`, and hitting Return. In the Command Prompt, use the following command to navigate to the `data` folder:

   ```
   cd /D %userprofile%\Desktop\intro-python\data
   ```

2. Start Jupyter server: **Unix shell**

```
jupyter notebook
```

Command Prompt (Windows)

```
python -m notebook
```

3. Launch the notebook by clicking on the "New" button on the right and selecting "Python 3" from the drop-down menu.

3.5.0.2 Anaconda Navigator

1. Launch Anaconda Navigator. It might ask you if you'd like to send anonymized usage information to Anaconda developers. Make your choice and click "Ok, and don't show again" button.
2. Find the "Notebook" tab and click on the "Launch" button. Anaconda will open a new browser window or tab with a Notebook Dashboard showing you the contents of your Home (or User) folder.
3. Navigate to the `data` directory by clicking on the directory names leading to it. `Desktop`, `intro-python`, then `data`:
4. Launch the notebook by clicking on the "New" button and then selecting "Python 3".

3.6 Option B: IPython interpreter

IPython is an alternative solution situated somewhere in between the plain-vanilla Python interpreter and Jupyter Notebook. It provides an interactive command-line based interpreter with various convenience features and commands. You should have IPython on your system if you installed Anaconda.

To start using IPython, execute:

```
ipython
```

3.7 Option C: plain-vanilla Python interpreter

To launch a plain-vanilla Python interpreter, execute:

```
python
```

If you are using Git Bash on Windows, you have to call Python *via* `winpty`:

```
winpty python
```

The instructions on this page were adapted from the setup instructions of the Software Carpentries "Programming with Python" course and their Workshop Template Python installation instructions, both released under the Creative Commons Attribution license. Changes to the material were made, and can be tracked in the Git repository associated with this course.

4 Course Materials

You can download the course materials as individual files or as zip files

- insert links to datasets and notebooks and zip files etc. below

Part II

Introduction

What is Python? Why do we like it so much more than R? Let's get going!

5 Introduction to python

5.1 Variables, values and their types

The cell below contains Python code that can be executed by the Python interpreter. One of the most basic things that we can do with Python is to use it as a calculator:

```
2+2
```

```
4
```

Great, but there are many calculators. It gets more interesting when we use **variables** to store information. This is done with the = operator. In Python, variable names: - can include letters, digits, and underscores - cannot start with a digit - are case sensitive.

```
x = 3.0
```

Once assigned, variables can be used in new operations:

```
y = 2.0
x + y
```

```
5.0
```

Python knows various types of data. Three common ones are:

- integer numbers
- floating point numbers
- strings

```
text = "Data Carpentry"
number = 42
pi_value = 3.14159265358
```

In the example above, three variables are assigned. Variable `number` is an integer number with a value of 42 while `pi_value` is a floating point number and `text` is of type string.

Using the `type` command, it is possible to check the data type of a variable.

```
text
```

```
'Data Carpentry'
```

```
type(text)
```

```
str
```

```
number
```

```
42
```

```
type(number)
```

```
int
```

```
pi_value
```

```
3.14159265358
```

```
type(pi_value)
```

```
float
```

5.2 Output versus printing

In the above examples, most of the times output is printed directly below the cell, but not always the output is printed and not all operations are printed. The `print` command can be used to control what is printed when.

Note, that text (strings) always has to be surrounded by `"` or `'`.

```
print("Hello World")
```

```
Hello World
```

In the example below we first print the value of the variable `number` using the `print` command, and then call the variable:

```
print(number)
number
```

```
42
```

```
42
```

Now we do it the other way around:

```
number
print(number)
```

```
42
```

When not using the `print` command, only the output of the last operation in the input cell is printed. If the last operation is the assignment of a variable, nothing will be printed.

In general `print` is the only way to print output to the screen when you are not working in an interactive environment like Jupyter (as we are doing now).

Rule of thumb: use the normal output for quick checking the output of an operation while developing in your Jupyter notebook, use `print` for printing output that still needs to be there in the future while your scripts get more complicated.

5.3 Exercise 0

5.4 Mathematical operations

In Python you can do a wide variety of mathematical operations. A few examples:

```
summing = 2 + 2
multiply = 2 * 7
power = 2 ** 16
modulo = 13 % 5

print("Sum: ", summing)
print("Multiply: ", multiply)
print("Power: ", power)
print("Modulo: ", modulo)
```

```
Sum:  4
Multiply:  14
Power:  65536
Modulo:  3
```

Once we have data stored in variables, we can use the variables to do calculations.

```
number = 42
pi_value = 3.14159265358

output = number * pi_value
print(output)
```

```
131.94689145036
```

5.5 Built-in Python functions

To carry out common tasks with data and variables in Python, the language provides us with several built-in functions. Examples of built-in functions that we already used above are `print` and `type`.

Calling a function When we want to make use of a function (referred to as calling the function), we type the name of the function followed by parentheses. Between the parentheses we can pass arguments.

Arguments We typically provide a function with 'arguments' to tell python which values or variables are used to perform the body of the function. In the example below `type` is the function name and `pi_value` is the argument.

```
type(pi_value)
```

```
float
```

Other useful built-in functions are `abs()`, `max()`, `min()`, `range()`. Find more built-in functions here.

```
max([1,2,3,2,1])
```

```
3
```

5.6 Boolean values, Logical expressions and operators

In programming you often need to know if something is `True` or `False`. `True` and `False` are called Boolean values and have their own data type (`bool` so they are not of type `str`!!). `True` and `False` are the only two Boolean values.

```
a = True
a
```

```
True
```

```
b = False
b
```

```
False
```

```
type(a)
```

```
bool
```

Comparison operators (e.g. `>`, `<`, `==`) are used in an expression to compare two values. The result of this expression is either `True` or `False`. Why this is useful we will show later (see if-statements).

```
3 > 4
```

```
False
```

`3 > 4` is an example of a 'logical expression' (also known as condition), where `>` is the comparison operator.

```
4 > 3
```

```
True
```

`==` is another comparison operator to check if two values or variables are the same. If this is the case it will return `True`

```
four = 4          # first we create a variable
four == 4         # then we check if it is equal to 4
```

```
True
```

`!=` is used to check if two values or variable are **not** the same. If this is the case it will return `True`

```
print("Four is not equal to 5: ", four != 5)
print("Four is not equal to 4: ", four != 4)
```

```
Four is not equal to 5:  True
Four is not equal to 4:  False
```

`and`, `or` and `not` are 'logical operators', and are used to join two logical expressions (or revert a logical expression in the case of `not`) to create more complex conditions.

`and` will return `True` if both expression on either side are `True`.

```
a = True
b = True
a and b
```

```
True
```

```
a = True
b = False
a and b
```

```
False
```

```
4 > 3 and 9 > 3
```

```
True
```

`or` is used to check if at least one of two logical expressions are `True`. If this is the case it will return `True`.

```
3 > 4 or 9 > 3
```

```
True
```

```
4 > 3 or 9 > 3
```

```
True
```

In the last three examples you can see that multiple expressions can be combined in a single line of Python code. Python evaluates the expressions one by one. `4 > 3` would return `True`, `9 > 3` would return `True`, so `4 > 3 or 9 > 3` would translate to `True or True`.

It is also possible to assign the output of an expression to a variable:

```
greater = 3 > 4
print("3 > 4 : ", greater)
```

```
3 > 4 :  False
```

The `not` operator can be used to reverse the Boolean value. If you apply `not` to an expression that evaluates to `True`, then you get `False` as a result. If you apply `not` to an expression that evaluates to `False`, then you get `True` as a result:

```
not 4 > 3
```

```
False
```

5.7 Exercise 1 to 3

5.8 The if-statement

If statements can be used to perform tasks only when a certain condition is met.

```
num = 101

if num > 100:
    print('number is greater than 100')
```

```
number is greater than 100
```

As you can see, the line `print(...` starts with 4 spaces indentation. In Python indentation is very important. Python uses indentation to determine which lines of code belong to what part of the code. This is mostly important when defining e.g. if-statements, for loops or functions. After the if condition, all lines with indentation are only performed when the if-condition is met.

```
num = 99
if num > 100:
    print('This line is only executed when num > 100')
    print('This line is only executed when num > 100')

    print('This line is only executed when num > 100')

print('This line is always executed')
```

```
This line is always executed
```

It is also possible to specify a task that is performed when the condition is not met using `else` (note the use of indentation):

```
num = 37

if num > 100:
    print('number is greater than 100')
else:
    print('number is not greater than 100')
```

```
print('done')
```

```
number is not greater than 100
done
```

An `if ... else` statement can be extended with (one or more) `elif` to specify more tasks that need to be performed on other conditions. These extended `if ... else` statements always start with `if` followed by (one or more) `elif`. When an `else` statement is included it is always the last statement.

Order matters: The statements (or conditions) are checked in order from top to bottom and only the task belonging to the first condition that is met is being performed.

```
num = -3

if num > 0:
    print(num, 'is positive')
elif num == 0:
    print(num, 'is zero')
else:
    print(num, 'is negative')
```

```
-3 is negative
```

Along with the `>` and `==` comparison operators that we have already used for comparing values in our logical expressions above, there are a few more options to know about:

- >: greater than
- <: less than
- ==: equal to
- !=: does not equal
- >=: greater than or equal to
- <=: less than or equal to

We can combine logical statements using `and` and `or` in more complex conditions in if statements.

```
if (1 < 0) or (1 >= 0):
    print('at least one the above logical statements is true')
```

```
at least one the above logical statements is true
```

While `and` is only true if both parts are true

```
if (1 < 0) and (1 >= 0):
    print('both tests are true')
else:
    print('at least one of both tests is not true')
```

```
at least one of both tests is not true
```

5.9 Lists and Tuples

Until now we have worked with values and variables that hold one value or string. Now we will go into other data types that can combine multiple values or strings.

Lists are common data structures to hold a sequence of elements. We can create a list by putting values inside square brackets and separating the values with commas.

```
numbers = [1, 2, 3]
print(numbers)
```

```
[1, 2, 3]
```

Each element can be accessed by an index. The index of the first element in a list in Python is 0 (in some other programming languages that would be 1).

```
print("The first element in the list numbers is: ", numbers[0])
```

```
The first element in the list numbers is:
```

```
1
```

```
type(numbers)
```

```
list
```

A total number of items in a list is called the 'length' and can be calculated using the `len()` function.

```
len(numbers)
```

```
3
```

You can do various things with lists. E.g. it is possible to sum the items in a list (when the items are all numbers)

```
print("The sum of the items in the list is:", sum(numbers))
print("The mean of the items in the list is:", sum(numbers)/len(numbers))
```

```
The sum of the items in the list is: 6
The mean of the items in the list is: 2.0
```

```
numbers[3]
```

```
IndexError: list index out of range
```

This error is expected. The list consists of three items, and the indices of those items are 0, 1 and 2.

```
numbers[-1]
```

```
3
```

Yes, we can use negative numbers as indices in Python. When we do so, the index -1 gives us the last element in the list, -2 the second to last, and so on. Because of this, numbers[2] and numbers[-1] point to the same element.

```
numbers[2] == numbers[-1]
```

```
True
```

It is also possible to combine strings in a list:

```
words = ["cat", "dog", "horse"]
words[1]
```

```
'dog'
```

```
type(words)
```

```
list
```

```
if type(words) == type(numbers):
    print("these variables have the same type!")
```

```
these variables have the same type!
```

It is also possible to combine values of different type (e.g. strings and integers) in a list

```
newlist = ["cat", 1, "horse"]
```

The type of the variable newlist is `list`. The elements of the list have their own data type:

```
type(newlist[0])
```

```
str
```

```
type(newlist[1])
```

```
int
```

It is possible to add numbers to an existing list using `list.append()`

```
numbers.append(4)
print(numbers)
```

```
[1, 2, 3, 4]
```

Using the index of an item, you can replace the item in a list:

```
numbers[2] = 333
print(numbers)
```

```
[1, 2, 333, 4]
```

A tuple is similar to a list in that it's a sequence of elements. However, tuples can not be changed once created (they are "immutable"). Tuples are created by placing comma-separated values inside parentheses `()` (instead of square brackets `[]`).

```
# Tuples use parentheses
a_tuple = (1, 2, 3)
another_tuple = ('blue', 'green', 'red')

# Note: lists use square brackets
a_list = [1, 2, 3]
```

```
a_list[1] = 5
print(a_list)
```

```
[1, 5, 3]
```

```
a_tuple[1] = 5
print(a_tuple)
```

```
TypeError: 'tuple' object does not support item assignment
```

Here we see that once the tuple is created, we cannot replace any of the values inside of the tuple.

```
type(a_tuple)
```

```
tuple
```

5.10 Dictionaries

A dictionary is another way to store multiple items into one object. In dictionaries, however, this is done with keys and values. This can be useful for several reasons, one example is to store model settings, parameters or variable values for multiple scenarios.

```
my_dict = {'one': 'first', 'two': 'second'}
my_dict
```

```
{'one': 'first', 'two': 'second'}
```

We can access dictionary items by their key:

```
my_dict['one']
```

```
'first'
```

And we can add new key-value pairs like that:

```
my_dict['third'] = 'three'
my_dict
```

```
{'one': 'first', 'two': 'second', 'third': 'three'}
```

Dictionary items are key-value pairs. The keys are changeable and unique. The values are changable, but not necessarily unique.

```
my_dict['third'] = 'three'
my_dict
```

```
{'one': 'first', 'two': 'second', 'third': 'three'}
```

```
print("Dictionary keys: ", my_dict.keys())
print("Dictionary values: ", my_dict.values())
print("Dictionary items (key, value): ", my_dict.items())
```

```
Dictionary keys:  dict_keys(['one', 'two', 'third'])
Dictionary values:  dict_values(['first', 'second', 'three'])
Dictionary items (key, value):  dict_items([('one', 'first'), ('two', 'second'), ('third', '
```

5.11 Exercise 4 to 7

5.12 For loops

Let's have a look at our list again. One way to print each number is to use three print statements:

```
numbers = [5, 6, 7]
print(numbers[0])
print(numbers[1])
print(numbers[2])
```

```
5
6
7
```

A more efficient (less typing) and reliable way to print each element of a list is to loop over the list using a for loop:

```
for item in numbers:
    print(item)
```

```
5
6
7
```

The improved version uses a for loop to repeat an operation — in this case, printing — once for each item in a sequence. Note that (similar to if statements) Python needs indentation (4 whitespaces) to determine which lines of code are part of the for loop.

If we want to also get the index, we can use the built-in function enumerate:

```
words = ["cat", "dog", "horse"]

for index, item in enumerate(words):
    print(index)
    print(item)
```

```
0
cat
1
dog
2
horse
```

For loops can also be used with dictionaries. Let's take our dictionary from the previous section and inspect the dictionary items

```
for item in my_dict.items():
    print(item, "is of type", type(item))
```

```
('one', 'first') is of type <class 'tuple'>
('two', 'second') is of type <class 'tuple'>
('third', 'three') is of type <class 'tuple'>
```

We can extract the keys and values from the items directly in the `for` statement:

```
for key, value in my_dict.items():
    print(key, "->", value)
```

```
one -> first
two -> second
third -> three
```

5.13 Functions

We have already seen some built-in functions: e.g. `print`, `type`, `len`. And we have seen special functions that belong to a variable (python object) like `my_dict.items()` and `my_list.append()`. There are more built-in functions e.g. for mathematical operations:

```
sum(numbers)
```

```
18
```

Please refer to https://docs.python.org/3/library/functions.html for more built-in functions.

5.13.1 Writing own functions

We will now turn to writing own functions. When should you write your own function?
1. If the functionality is not covered by an out-of-the-box function like the built-in functions or another python package
2. When code is getting pretty long, you can split it up into logical and reusable units
3. When code is often reused, e.g. you are reading in tens of spreadsheets and you need to clean them all in the same way. Instead of typing the line of code over and over again, it is more elegant and looks cleaner to create a function.

4. When code may be reused outside your current project. Scripts and the functions in a script can be imported in other scripts to be able to reuse them.

A big advantage of not having duplicate code inside your script or in multiple scripts is that when you want to make a slight modification to a function, you only have to do this modification in one place, instead of multiple lines that are doing more or less similar things.

Python provides for this by letting us define things called 'functions'. Let's start by defining a function fahr_to_celsius that converts temperatures from Fahrenheit to Celsius:

```
def fahr_to_celsius(temp_fahrenheit):
    temp_celsius = (temp_fahrenheit - 32) * (5/9)
    return temp_celsius
```

The function definition opens with the keyword `def` followed by the name of the function `fahr_to_celsius` and a parenthesized list of variables (in this case only one `temp_fahrenheit`). The body of the function — the statements that are executed when it runs — is indented below the definition line. The body concludes with a `return` keyword followed by the return value.

When we call the function, the values we pass to it as arguments are assigned to the variables in the function definition so that we can use them inside the function. Inside the function, we use a return statement to send a result back to whoever asked for it.

Let's try running our function.

```
fahr_to_celsius(98)
```

```
36.66666666666667
```

```
print('freezing point of water:', fahr_to_celsius(32), 'C')
print('boiling point of water:', fahr_to_celsius(212), 'C')
```

```
freezing point of water: 0.0 C
boiling point of water: 100.0 C
```

Here we directly passed a value to the function. We can also call the function with a variable:

```
a = 0
print(fahr_to_celsius(a))
```

```
-17.77777777777778
```

What happens if you pass a variable name that is not defined yet?

```
print(fahr_to_celsius(b))
```

```
-17.77777777777778
```

5.14 Exercise 8 and 9

Part III

Data Science with Pandas

Let's try out the Pandas library!

```
2 + 2
```

```
4
```

```
print("Hello World")
```

```
Hello World
```

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.